

2

AD-A204 279

AVF Control Number: AVF-VSR-018  
SZT-AVF-018

Ada COMPILER  
VALIDATION SUMMARY REPORT:  
Certificate Number: 88052411.09118  
SYSTEM KG  
SYSTEM Ada Compiler VAX/VMS  
Version 1.8  
VAX 8530

Completion of On-Site Testing:  
88-05-24

Prepared By:  
IABG m.b.H., Dept SZT  
Einsteinstrasse 20  
8012 Ottobrunn  
Federal Republic of Germany

Prepared For:  
Ada Joint Program Office  
United States Department of Defense  
Washington, D.C. 20301-3081

DTIC  
ELECTE  
S FEB 14 1989 D  
SH

Ada is a registered trademark of the United States Government  
(Ada Joint Program Office).

DISTRIBUTION STATEMENT A

Approved for public release;  
Distribution Unlimited

89 2 13 091

# UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report; SYSTEAM KG, SYSTEAM Ada Compiler, VAX/VMS, Version 1.8, VAX 8530 (Host and Target). (880524 IL 09118)		5. TYPE OF REPORT & PERIOD COVERED 24 May 1988 to 24 May 1989
7. AUTHOR(s) IABG, Ottobrunn, Federal Republic of Germany.		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION AND ADDRESS IABG, Ottobrunn, Federal Republic of Germany.		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) IABG, Ottobrunn, Federal Republic of Germany.		12. REPORT DATE 24 May 1988
		13. NUMBER OF PAGES 49 p.
		15. SECURITY CLASS (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report)  Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)  UNCLASSIFIED		
18. SUPPLEMENTARY NOTES		
19. KEYWORDS (Continue on reverse side if necessary and identify by block number)  Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD- 1815A, Ada Joint Program Office, AJPO		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) SYSTEAM Ada Compiler VAX/VMS, Version 1.8, SYSTEAM KG, IABG, VAX 8530 under VMS, Version 4.7 (Host and (Target), ACVC 1.9.		

DD FORM

1473

EDITION OF 1 NOV 65 IS OBSOLETE

1 JAN 73

S/N 0102-LF-014-8661

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

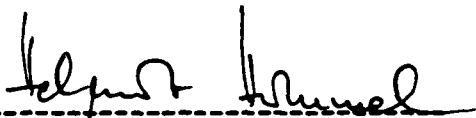
Compiler Name: SYSTEAM Ada Compiler VAX/VMS  
Compiler Version: Version 1.8

Certificate Number: 880524I1.09118

Host and target: VAX 8530 under VMS, Version 4.7

Testing Completed 88-05-24 Using ACVC 1.9

This report has been reviewed and is approved.



IABG m.b.H., Dept SZT  
Dr. H. Hummel  
Einsteinstrasse 20  
8012 Ottobrunn  
Federal Republic of Germany



Ada Validation Organization  
Dr. John F. Kramer  
Institute for Defense Analyses  
Alexandria VA 22311



Ada Joint Program Office  
Virginia L. Castor  
Director  
Department of Defense  
Washington DC 20301

# CONTENTS

## CHAPTER 1 INTRODUCTION

1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT . . . . .	1-2
1.2	USE OF THIS VALIDATION SUMMARY REPORT . . . . .	1-2
1.3	REFERENCES . . . . .	1-3
1.4	DEFINITION OF TERMS . . . . .	1-4
1.5	ACVC TEST CLASSES . . . . .	1-5

## CHAPTER 2 CONFIGURATION INFORMATION

2.1	CONFIGURATION TESTED . . . . .	2-1
2.2	IMPLEMENTATION CHARACTERISTICS . . . . .	2-1

## CHAPTER 3 TEST INFORMATION

3.1	TEST RESULTS . . . . .	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS . . . . .	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER . . . . .	3-2
3.4	WITHDRAWN TESTS . . . . .	3-2
3.5	INAPPLICABLE TESTS . . . . .	3-2
3.6	TEST, PROCESSING, AND EVALUATION MODIFICATIONS . . . . .	3-4
3.7	ADDITIONAL TESTING INFORMATION . . . . .	3-4
3.7.1	Prevalidation . . . . .	3-4
3.7.2	Test Method . . . . .	3-4
3.7.3	Test Site . . . . .	3-5

## APPENDIX A DECLARATION OF CONFORMANCE

## APPENDIX B APPENDIX F OF THE Ada STANDARD

## APPENDIX C TEST PARAMETERS

## APPENDIX D WITHDRAWN TESTS



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

## CHAPTER 1

### INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

(KR)

## INTRODUCTION

### 1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- . To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 88-05-24 at SYSTEAM KG at Karlsruhe.

### 1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse  
Ada Joint Program Office  
OUSDRE  
The Pentagon, Rm 3D-139 (Fern Street)  
Washington DC 20301-3081

or from:

IABG m.b.H., Dept SZT  
Einsteinstrasse 20  
8012 Ottobrunn  
Federal Republic of Germany

## INTRODUCTION

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization  
Institute for Defense Analyses  
1801 North Beauregard Street  
Alexandria VA 22311

### 1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

## INTRODUCTION

### 1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect



because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

## 1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

## INTRODUCTION

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK\_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK\_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK\_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

## CHAPTER 2

### CONFIGURATION INFORMATION

#### 2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: SYSTEAM Ada Compiler VAX/VMS, Version 1.8

ACVC Version: 1.9

Certificate Number: BB0524I1.09118

Host and Target Computer:

Machine: VAX 8530

Operating System: VMS Version 4.7

Memory Size: 32 MB

#### 2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

## CONFIGURATION INFORMATION

### . Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

### . Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed `SYSTEM.MAX_INT`. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

### . Predefined types.

This implementation supports the additional predefined types `SHORT_INTEGER`, `SHORT_SHORT_INTEGER`, `SHORT_FLOAT`, `LONG_FLOAT` and `LONG_LONG_FLOAT` in the package `STANDARD`. (See tests B86001C and B86001D.)

### . Based literals.

An implementation is allowed to reject a based literal with a value exceeding `SYSTEM.MAX_INT` during compilation, or it may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` during execution. This implementation raises `CONSTRAINT_ERROR` during execution. (See test E24101A.)

### . Expression evaluation.

Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

No exception is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

## CONFIGURATION INFORMATION

No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

### . Rounding.

The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

### . Array types.

An implementation is allowed to raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` for an array having a `'LENGTH` that exceeds `STANDARD.INTEGER'LAST` and/or `SYSTEM.MAX_INT`. For this implementation:

Declaration of an array type or subtype declaration with more than `SYSTEM.MAX_INT` components raises no exception. (See test C36003A.)

No exception is raised when `'LENGTH` is applied to an array type with `INTEGER'LAST + 2` components. (See test C36202A.)

No exception is raised when `'LENGTH` is applied to an array type with `SYSTEM.MAX_INT + 2` components. (See test C36202B.)

A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` raises `CONSTRAINT_ERROR` when the array type is declared. (See test C52103X.)

A packed two-dimensional `BOOLEAN` array with more than `INTEGER'LAST` components raises `CONSTRAINT_ERROR` when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than `INTEGER'LAST` may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises `CONSTRAINT_ERROR` when the array type is declared. (See test E52103Y.)

## CONFIGURATION INFORMATION

In assigning one-dimensional array types, the expression does not appear to be evaluated in its entirety before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

### . Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression does not appear to be evaluated in its entirety before `CONSTRAINT_ERROR` is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

### . Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before `CONSTRAINT_ERROR` is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

### . Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

## CONFIGURATION INFORMATION

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE\_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE\_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

There are restrictions for alignment clauses within record representation clauses. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

### . Pragma.

The pragma INLINE is supported for procedures. The pragma INLINE is supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

### . Input/output.

The package SEQUENTIAL\_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT\_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes IN\_FILE and OUT\_FILE are supported for SEQUENTIAL\_IO. (See tests CE2102D and CE2102E.)

## CONFIGURATION INFORMATION

Modes IN\_FILE, OUT\_FILE, and INOUT\_FILE are supported for DIRECT\_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL\_IO and DIRECT\_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL\_IO and DIRECT\_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to the last element. (See test CE2208B.)

An existing text file can be opened in OUT\_FILE mode, can be created in OUT\_FILE mode, and can be created in IN\_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for both reading only. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each non-temporary external file for sequential I/O for reading only. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each non-temporary external file for direct I/O for reading only. (See tests CE2107F..I (4 tests), CE2110B, and CE2111H.)

An internal sequential access file and an internal direct access file cannot be associated with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file cannot be deleted but closed for SEQUENTIAL\_IO, DIRECT\_IO, and TEXT\_IO. (See test CE2110B.)

Temporary sequential files are not given names. Temporary direct files are not given names. (See tests CE2108A and CE2108C.)

### . Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)



## CHAPTER 3

### TEST INFORMATION

#### 3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 81 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. Modifications to the code, processing, or grading for 13 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

#### 3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
-----	_A_	_B_	_C_	_D_	_E_	_L_	-----
Passed	109	1049	1776	17	17	46	3014
Inapplicable	1	2	77	0	1	0	81
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

## TEST INFORMATION

### 3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER														TOTAL
	__2__	__3__	__4__	__5__	__6__	__7__	__8__	__9__	__10__	__11__	__12__	__13__	__14__	_____	
Passed	182	569	644	245	166	98	142	326	137	36	234	3	232	3014	
Inapplicable	22	3	30	3	0	0	1	1	0	0	0	0	21	81	
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27	
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122	

### 3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35904R	C35A03E	C35A03R	C37213H
C37213J	C37215C	C37215E	C37215G	C37215H
C38102C	C41402A	C45332A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A
CE2401H	CE3208A			

See Appendix D for the reason that each of these tests was withdrawn.

### 3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 81 tests were inapplicable for the reasons indicated:

- Tests C24113D..Y (22 tests) and C45621Q..Z (10 tests) contain lines of lengths greater than 80 characters which is not supported by this compiler.

## TEST INFORMATION

- . A39005G uses an alignment clause with an alignment of 8 within a record representation clause.
- . C34007P, C34007S are expected to raise CONSTRAINT\_ERROR. this implementation optimizes the code at compile time on lines 201 and 217 respectively, thus avoiding the operation which would raise CONSTRAINT\_ERROR and so no exception is raised. The AVO ruled this behavior acceptable and the test NA.
- . C41401A is expected to raise CONSTRAINT\_ERROR for the evaluation of certain attributes, however this implementation derives the values from the subtype of the prefix at compile time, as allowed by 11.6(7) LRM. Therefore elaboration of the prefix is not involved and CONSTRAINT\_ERROR is not raised. The AVO ruled this behavior acceptable and the test NA.
- . The following tests use LONG\_INTEGER, which is not supported by this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

- . C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.
- . C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.
- . C47004A is expected to raise CONSTRAINT\_ERROR whilst evaluating the comparison on line 51, but this compiler evaluates the result without invoking the basic operation qualifier (as allowed by 11.6(7) LRM) which would raise CONSTRAINT\_ERROR and so no exception is raised. The AVO ruled this behavior acceptable and the test NA.
- . C86001F redefines package SYSTEM, but TEXT\_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT\_IO.
- . C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.
- . CE2108A, CE2108C, CE3112A are inapplicable because temporary files (sequential, direct, text) do not have names.
- . CE2107B..E (4 tests), CE2107G..I (3 tests), CE2111D, CE2111H, CE2110B, CE3111B..E (4 tests), CE3114B, CE3115A and CE3108A are inapplicable because multiple internal files cannot be associated with the same external file. The proper exception is raised when multiple access is attempted.

## TEST INFORMATION

- . EE2401D uses instantiations of the package DIRECT\_IO with unconstrained array types. This instantiation is rejected by this compiler.

### 3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 13 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B24009A	B29001A	B38003A	B38009A
B38009B	B51001A	B91001H	BC2001D	BC2001E
BC3204B	BC3205B	BC3205D		

### 3.7 ADDITIONAL TESTING INFORMATION

#### 3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the SYSTEAM Ada Compiler VAX/VMS Version 1.8 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

#### 3.7.2 Test Method

Testing of the SYSTEAM Ada Compiler VAX/VMS Version 1.8 using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a VAX 8530 host operating under VMS, Version 4.7.

## TEST INFORMATION

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled on the VAX 8530. Object files were linked and executed on the target. Results were printed from the host computer.

The compiler was tested using command scripts provided by SYSTEAM KG and reviewed by the validation team. The compiler was tested using all default settings.

Tests were compiled, linked, and executed (as appropriate) using a single host computer. Test output, compilation listings of Class B tests and tests that raised an error during compilation, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

### 3.7.3 Test Site

Testing was conducted at SYSTEAM KG at Karlsruhe and was completed on 88-05-24.

APPENDIX A  
DECLARATION OF CONFORMANCE

SYSTEM KG has submitted the following Declaration of  
Conformance concerning the SYSTEM Ada Compiler  
VAX/VMS, Version 1.8.

## DECLARATION OF CONFORMANCE

### DECLARATION OF CONFORMANCE

Compiler Implementor: SYSTEAM KG  
Ada Validation Facility: IABG m.b.H., Dept SZT  
Ada Compiler Validation Capability (ACVC) Version: 1.9

#### Base Configuration

Base Compiler Name: SYSTEAM Ada Compiler VAX/VMS  
Base Compiler Version: Version 1.8  
Host and Target Architecture ISA: VAX 8530 VMS 4.7

#### Implementor's Declaration

I, the undersigned, representing SYSTEAM KG, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that SYSTEAM KG is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name:



Date: 24 May 1988

-----  
SYSTEAM KG  
Dr. Winterstein,

#### Owner's Declaration

I, the undersigned, representing SYSTEAM KG, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.



Date: 24 May 1988

-----  
SYSTEAM KG  
Dr. Winterstein,

## APPENDIX B

### APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the SYSTEAM Ada Compiler VAX/VMS, Version 1.8, are described in the following sections, which discuss topics in Appendix F of the Ada Standard. They are taken from the vendor's User Manual. Implementation-specific portions of the package STANDARD are also included in this appendix.



The specification of the package STANDARD is outlined here; it contains all predefined identifiers of the VAX/VMS implementation.

The operations defined for the predefined types are not mentioned since they are implicitly declared according to the language rules. Anonymous types (such as UNIVERSAL\_INTEGER) are not mentioned either.

#### PACKAGE standard IS

TYPE boolean IS (false, true);

TYPE short\_short\_integer IS RANGE - 128 .. 127;

TYPE short\_integer IS RANGE - 32\_768 .. 32\_767;

TYPE integer IS RANGE - 2\_147\_483\_648 .. 2\_147\_483\_647;

TYPE short\_float IS DIGITS 6 RANGE

- hex16#0.7FFF\_FF8#E+32 .. 16#0.7FFF\_FF8#E+32;

-- the corresponding machine type is F-FLOAT

TYPE float IS DIGITS 9 RANGE

- 16#0.7FFF\_FFFF\_FFFF\_FF8#E+32 ..

16#0.7FFF\_FFFF\_FFFF\_FF8#E+32;

-- the corresponding machine type is D-FLOAT

TYPE long\_float IS DIGITS 15 RANGE

- 16#0.7FFF\_FFFF\_FFFF\_FC#E+256 ..

16#0.7FFF\_FFFF\_FFFF\_FC#E+256;

-- the corresponding machine type is G-FLOAT

TYPE long\_long\_float IS DIGITS 33 RANGE

- 16#0.7FFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_C#E+4096 ..

16#0.7FFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_C#E+4096;

-- the corresponding machine type is H-FLOAT

-- TYPE character IS ... as in [ADA,Appendix C]

-- FOR character USE ... as in [ADA,Appendix C]

-- PACKAGE ascii IS ... as in [ADA,Appendix C]

-- Predefined subtypes and string types ... as in [ADA,Appendix C]

TYPE duration IS DELTA 2#1.0#E-14 RANGE

- 131\_072.0 .. 131\_071.999\_938\_964\_843.75;

## 15 Appendix F

This is the Appendix F required in [ADA], in which all implementation-dependent characteristics of an Ada implementation are described.

### F.1 Implementation-dependent pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

#### *F.1.1 Predefined language pragmas*

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here. All the other pragmas listed in Appendix B of [ADA] are implemented and have the effect described there.

**CONTROLLED**  
has no effect.

**INLINE**  
Inline expansion of subprograms is supported with following restrictions:  
the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

**INTERFACE**  
is only implemented for Assembly language; see §13.9 of this manual for details.

**MEMORY\_SIZE**  
has no effect.

**OPTIMIZE**  
has no effect.

**PACK**  
see §13.1.

**PRIORITY**

There are two implementation-defined aspects of this pragma: First, the range of the subtype **PRIORITY**, and second, the effect on scheduling (§5) of not giving this pragma for a task or main program. The range of subtype **PRIORITY** is 0 .. 15, as declared in the predefined library package **SYSTEM** (see §F.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving pragma **PRIORITY** for it is the same as if pragma **PRIORITY 0** had been given (i.e. the task has the lowest priority). Moreover, in this implementation the package **SYSTEM** must be named by a with clause of a compilation unit if the predefined pragma **PRIORITY** is used within that unit.

**SHARED**

has no effect. Note, however, that the implementation of tasking is such that every variable is treated as if pragma **SHARED** had been given for it.

**STORAGE\_UNIT**

has no effect.

**SUPPRESS**

has no effect, but see §F.1.2 for the implementation-defined pragma **SUPPRESS\_ALL**.

**SYSTEM\_NAME**

has no effect.

*F.1.2 Implementation-defined pragmas***SQUEEZE**

see §13.1.

**SUPPRESS\_ALL**

causes all the run-time checks described in [ADA,§11.7] except **ELABORATION\_CHECK** to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

## F.2 Implementation-dependent attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this chapter.

### *F.2.1 Language-defined attributes*

The name and type of all the language-defined attributes are as given in [ADA]. We note here only the implementation-dependent aspects.

#### ADDRESS

The value delivered by this attribute applied to an object is the address of the storage unit where this object starts.

For any other entity this attribute is not supported and will return the value `SYSTEM.null_address`.

#### SIZE

The only implementation-dependent aspect is as follows:

for an object of an unconstrained record type the value delivered by this attribute depends on the actual constrained, i.e. for different constraints the attribute will possibly yield different values.

#### STORAGE\_SIZE

The value delivered by this attribute applied to an access type is as follows:

If a length specification (`STORAGE_SIZE`, see §13.2) has been given for that type (static collection), the attribute delivers that specified value.

In case of a dynamic collection, i.e. no length specification by `STORAGE_SIZE` given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.

If the collection manager (cf. Chapter 12) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:

If a length specification (`STORAGE_SIZE`, see §13.2) has been given for the task type, the attribute delivers that specified value; the default value is returned elsewhere.

*F.2.2 Implementation-defined attributes*

There are no implementation-defined attributes.

### F.3 Specification of the package SYSTEM

The package SYSTEM of ([ADA,§13.7]) is reprinted here with all implementation-dependent characteristics and extensions filled in.

PACKAGE system IS

TYPE address IS PRIVATE;  
TYPE name IS (vax\_vms);

system\_name : CONSTANT name := vax\_vms;  
storage\_unit : CONSTANT := 8;  
memory\_size : CONSTANT := 2 \*\* 31;  
min\_int : CONSTANT := - 2\_147\_483\_648;  
max\_int : CONSTANT := 2\_147\_483\_647;  
max\_digits : CONSTANT := 33;  
max\_mantissa : CONSTANT := 31;  
fine\_delta : CONSTANT := 2#1.0#E-31;  
tick : CONSTANT := 0.2E-6;

SUBTYPE priority IS integer RANGE 0 .. 15;  
SUBTYPE external\_address IS string;  
SUBTYPE byte IS integer RANGE 0..255;  
TYPE long\_word IS ARRAY (0..3) OF byte;  
PRAGMA PACK (long\_word);

FUNCTION convert\_address  
    (addr : external\_address) RETURN address;

FUNCTION convert\_address  
    (addr : address) RETURN external\_address;

FUNCTION convert\_address  
    (addr : long\_word) RETURN address;

FUNCTION convert\_address  
    (addr : address) RETURN long\_word;

FUNCTION "+" (addr : address; offset : integer)  
    RETURN address;

PRIVATE

-- private declarations  
END system;

**F.4 Restrictions on representation clauses**

See §§13.2-13.5 of this manual.

**F.5 Conventions for implementation-generated names**

There are no implementation-generated names denoting implementation-dependent components ([ADA,§13.4]).

**F.6 Expressions in address clauses**

Address clauses ([ADA,§13.5]) are supported only for objects except for task objects. The object starts at the given address.

**F.7 Restrictions on unchecked conversions**

The implementation does support unchecked type conversions for all kind of source- and target-types with the restriction that the target type must not be an unconstrained array type. Note that if

target\_type'size > source\_type'size,

the result value of the unchecked conversion is unpredictable.

**F.8 Characteristics of the input-output packages**

The implementation-dependent characteristics of the input-output packages as defined in Chapter 14 of [ADA] are reported in Chapter 14 of this manual.

## 14 Input-output

In this chapter we follow the section numbering of Chapter 14 of [ADA] and provide notes for the use of the features described in each section.

### 14.1 External files and file objects

The total number of open files (including the two standard files) must not exceed 18. Any attempt to exceed this limit raises the exception `USE_ERROR`.

The only form of file sharing which is allowed is shared reading. If two or more files are associated with the same external file at one time (regardless of whether these files are declared in the same program or task), all of these (internal) files must be opened with the mode `IN_FILE`. An attempt to open one of these files with a mode other than `IN_FILE` will raise the exception `USE_ERROR`.

Files associated with terminal devices (which is only legal for text files) are excepted from this restriction. Such files may be opened with an arbitrary mode at the same time and associated with the same terminal device.

The following restrictions apply to the generic actual parameter for `ELEMENT_TYPE`:

- input/output of access types is not defined.
- input/output of unconstrained array types is only possible with a variable record format.
- for RMS sequential [relative or indexed] files the size of an object to be input or output must not be greater than 32767 [16383].
- input/output is not possible for an object whose (sub)type has a size which is not a multiple of `SYSTEM.STORAGE_UNIT`. Such objects may only exist for types for which a representation clause or pragma `SQUEEZE` is given. `USE_ERROR` will be raised by any attempt to read or write such an object or to open or create a file for such a (sub)type.

### 14.2 Sequential and direct files

Sequential and direct files are represented by RMS sequential, relative or indexed files with fixed-length or variable-length records. Each element of the file is stored in one record.



### 14.2.1 File management

Since there is a lot to say about this section, we shall introduce subsection numbers which do not exist in [ADA].

#### 14.2.1.1 The NAME and FORM parameters

The NAME parameter string must be a VMS file specification string and must not contain wild cards, even if that would specify a unique file. The function NAME will return a file specification string (including version number) which is the file name of the file opened or created.

The Syntax of the FORM parameter string is defined by:

```
form_parameter ::= [ form_specification { , form_specification } ]
```

```
form_specification ::= keyword [ => value ]
```

```
keyword ::= identifier
```

```
value ::= identifier | string_literal | numeric_literal
```

For identifier, numeric\_literal, string\_literal see [ADA,Appendix E]. Only an integer literal is allowed as numeric\_literal (see [ADA,§2.4]).

In the following, the form specifications which are allowed for all files are described.

```
ALLOCATION => numeric_literal
```

This value specifies the number of blocks which are allocated initially; it is only used in a create operation and ignored in an open operation. The value of ALLOCATION in the form string returned by the function FORM specifies the initial allocation size for existing files too.

```
EXTENSION => numeric_literal
```

This value specifies the number of blocks by which a file is extended if necessary; the value 0 means that the RMS default value is taken. For existing files, this value is only used for processing between an open and a close operation.

For details see the VAX-11 / RMS Reference Manual.

`MAX_RECORD_SIZE => numeric_literal`

This value specifies the maximum record size in bytes. The value 0 indicates that there is no limit; for direct files, this value is only allowed for indexed files, whereas for sequential files there is no such restriction. This form specification is only allowed for files with variable record format. If the value is specified for an existing file it must agree with the value of the external file.

For files with fixed-length records, the maximum record size equals `ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT`. If a fixed record format is used, all objects written to a file which are shorter than the maximum record size are filled up with zeros (`ASCII.NUL`).

`RECORD_FORMAT => VARIABLE | FIXED`

This form specification is used to specify the record format. If the format is specified for an existing file it must agree with the format of the external file.

### 14.2.1.2 Sequential files

A sequential file is represented by a RMS sequential file with either fixed-length or variable-length records which may be specified by the form parameter.

If a fixed record format is used, all objects written to a file which are shorter than the maximum record size are filled up with zeros (ASCII.NUL).

#### END\_OF\_FILE

If the keyword `END_OF_FILE` is specified for an existing file in an open for an output file, then the file is opened at the end of the file; i.e. the existing file is extended and not rewritten. This keyword is only allowed for an output file; it only has an effect in an open operation and is ignored in a create.

The default form string for a sequential file is :

```
"ALLOCATION      => 3,          EXTENSION      => 0, " &  
"RECORD_FORMAT => VARIABLE, MAX_RECORD_SIZE => 0  "
```

The default form may be used for all types (except for those excluded in §14.1).

### 14.2.1.3 Direct files

The implementation dependent type `COUNT` defined in the package specification of `DIRECT_IO` has an upper bound of :

```
COUNT'LAST = 2.147.483.647 (= INTEGER'LAST)
```

Direct files are represented by RMS sequential files with fixed-length records or by relative or indexed files with either fixed-length or variable-length records. For indexed files, the record index is stored as unsigned four bytes binary value in the first four bytes of each record. If not explicitly specified, the maximum record size equals `ELEMENT.TYPE'SIZE / SYSTEM.STORAGE_UNIT`.

```
BUCKET_SIZE => numeric_literal
```

This value specifies the number of blocks (one block is 512 bytes) for one bucket; the value 0 means that the value is evaluated by RMS to the minimal number of blocks which is necessary to contain one record. The value must be in the range from 0 up to 32. This form specification is only allowed for relative or indexed files. If the value is specified for an existing file it must agree with the value of the external file.

ORGANIZATION => INDEXED | RELATIVE | SEQUENTIAL

This form specification is used to specify the file organization. If the organization is specified for an existing file it must agree with the organization of the external file.

The default form string for a direct file is :

"ALLOCATION => 3. EXTENSION => 0. " &  
"ORGANIZATION => SEQUENTIAL, RECORD\_FORMAT => FIXED"

Indexed files with variable-length records and a maximum record size of 0 may be used for all types (except for those excluded in §14.1). Relative files with variable-length records may also be used for all types, but in this case a maximum record size must be specified explicitly. Sequential, relative or indexed files with fixed-length records may not be used for unconstrained array types.

### 14.3 Text input/output

Text files are represented as sequential files with variable record format. One line is represented as a sequence of one or more records; all records except for the last one have a length of exactly `MAX_RECORD_SIZE` and a continuation marker (`ASCII.LF`) at the last position. A line of length `MAX_RECORD_SIZE` is represented by one record of this length. A line terminator is not represented explicitly in the external file; the end of a record which is shorter than `MAX_RECORD_SIZE` or which has length exactly `MAX_RECORD_SIZE` and does not have a continuation marker as its last character is taken as a line terminator.

The value `MAX_RECORD_SIZE` may be specified by the form string for an output file and it is taken from the external file for an input file; for an input file, the value 0 stands for the default of 255. For all files which are created, the value `MAX_RECORD_SIZE` is used for the file attribute `MRS` (maximum record size).

A page terminator is represented as a record consisting of a single `ASCII.FF`. A record of length zero is assumed to precede a page terminator if the record before the page terminator is another page terminator or a record of length `MAX_RECORD_SIZE` with a continuation marker at the last position; this implies that a page terminator is preceded by a line terminator in all cases.

A file terminator is not represented explicitly in the external file; the end of the file is taken as a file terminator. A page terminator is assumed to precede the end of the file if there is not explicitly one as the last record of the file. For input from a terminal, a file terminator is represented as `ASCII.SUB` (= `CTRL/Z`).

### *14.3.1 File management*

In the following, the form specifications which are only allowed for text files or have a special meaning for text files are described.

#### **CHARACTER\_IO**

The predefined package **TEXT\_IO** was designed for sequential text files; moreover, this implementation always uses sequential files with a record structure, even for terminal devices. It therefore offers no language-defined facilities for modifying data previously written to the terminal (e.g. changing characters in a text which is already on the terminal screen) or for outputting characters to the terminal without following them by a line terminator. It also has no language-defined provision for input of single characters from the terminal (as opposed to lines, which must end with a line terminator, so that in order to input one character the user must type in that character and then a line terminator) or for suppressing the echo on the terminal of characters typed in at the keyboard.

For these reasons, in addition to the input/output facilities with record structured external files, another form of input/output is provided for text files: It is possible to transfer single characters from/to a terminal device. This form of input/output is specified by the keyword **CHARACTER\_IO** in the form string. If character i/o is specified, no other form specification is allowed and the file name must denote a terminal device.

For an infile, the external file (associated with a terminal) is considered to contain a single line. An **ASCII.SUB** (= **CTRL/Z**) character represents a line terminator followed by a page terminator followed by a file terminator. Arbitrary characters (including all control characters except for **ASCII.SUB**) may be read; a character read is not echoed to the terminal.

For an outfile, arbitrary characters (including all control characters and escape sequences) may be written on the external file (terminal). A line terminator is represented as **ASCII.CR** followed by **ASCII.LF**, a page terminator is represented as **ASCII.FF** and a file terminator is not represented on the external file.

Only for input files :

PROMPTING => string\_literal

This string is output on the terminal before an input record is read if the input file is associated with a terminal; otherwise this form specification is ignored.

Only for output files :

MAX\_RECORD\_SIZE => numeric\_literal

This value specifies the maximum length of a record in the external file. Each record which is not the last record of a line has exactly this maximum record size, with a continuation marker (ASCII.LF) at the last position. The value must be in the range 2 .. 255. If a file is created, the specified value (or the default of 255) is used for the file attribute MRS (maximum record size) of the external file. If the value is specified for an existing file it must be identical to the value of the external file.

The default form string for an input text file is :

"ALLOCATION => 3, EXTENSION => 0, PROMPTING => "" "" "

The default form string for an output text file is :

"ALLOCATION => 3, EXTENSION => 0, MAX\_RECORD\_SIZE => 255"

### *14.3.2 Default input and output files*

The standard input (resp. output) file is associated with the system default logical names SYS\$INPUT (resp. SYS\$OUTPUT) of VMS. If a program reads from the standard input file, the logical name SYS\$INPUT must denote an existing file. If a

program writes to the standard output file, a file with the logical name SYS\$OUTPUT is created if no such file exists; otherwise the existing file is extended.

The qualifiers /INPUT and /OUTPUT may be used for the VMS RUN command to associate VMS files with the standard files of TEXT\_IO.

The name and form strings for the standard files are :

```
standard_input :  NAME => "SYS$INPUT:"  
                  FORM => "PROMPTING => "*" " "  
  
standard_output : NAME => "SYS$OUTPUT:"  
                  FORM => "MAX_RECORD_SIZE => 255"
```

#### *14.9.10 Implementation-defined types*

The implementation dependent types COUNT and FIELD defined in the package specification of TEXT\_IO have the following upper bounds :

COUNT'LAST = 2\_147\_483\_647 (= INTEGER'LAST)

FIELD'LAST = 255



### 14.4 Exceptions in input-output

For each of `NAME_ERROR`, `USE_ERROR`, `DEVICE_ERROR` and `DATA_ERROR` we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package `IO_EXCEPTIONS` can be raised are as described in [ADA,§14.4].

#### `NAME_ERROR`

- in an `OPEN` operation, if the specified file does not exist;
- in a `CREATE` operation, if the `NAME` string contains an explicit version number and the specified file already exists;
- if the name parameter in a call of the `CREATE` or `OPEN` procedure is not a legal VMS file specification string; for example, if it contains illegal characters, is too long or is syntactically incorrect; and also if it contains wild cards, even if that would specify a unique file.

#### `USE_ERROR`

- if an attempt is made to increase the total number of open files (including the two standard files) to more than 18;
- whenever an error occurred during an operation of the underlying RMS system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a size restriction is violated, a capacity limit is exceeded or for similar reasons;
- if the function `NAME` is applied to a temporary file;
- if the characteristics of the external file are not appropriate for the file type; for example, if the record size of a file with fixed-length records does not correspond to the size of the element type of a `DIRECT_IO` or `SEQUENTIAL_IO` file. In general it is only guaranteed that a file which is created by an Ada program may be reopened by another program if the file types and the form strings are the same;
- if two or more (internal) files are associated with the same external file at one time (regardless of whether these files are declared in the same program or task), and an attempt is made to open one of these files with mode other than `IN_FILE`. However, files associated with terminal devices (which is only legal for text files) are excepted from this restriction. Such files may be opened with an arbitrary mode at the same time and associated with the same terminal device;
- if a given `FORM` parameter string does not have the correct syntax or if a condition on an individual form specification described in §§14.2-3 is not fulfilled;
- if an attempt is made to open or create a sequential or direct file for an element type whose size is not a multiple of `SYSTEM.STORAGE_UNIT`; or if an attempt is made to read or write an object whose (sub)type has a size which is not a

DEVICE\_ERROR

DATA\_ERROR

- ## 14.6 Low level input-output

PACKAGE low\_level\_io IS

```
TYPE data_type IS
    RECORD
        NULL;
    END RECORD;
```

```
PROCEDURE receive_control (device : device_type;
                           data   : IN OUT data_type);
```

END low\_level\_io;

Note that the enumeration type `DEVICE_TYPE` has only one enumeration value, `NULL_DEVICE`; thus the procedures `SEND_CONTROL` and `RECEIVE_CONTROL` can be called, but `SEND_CONTROL` will have no effect on any physical device and the value of the actual parameter `DATA` after a call of `RECEIVE_CONTROL` will have no physical significance.

## APPENDIX C

### TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name_and_Meaning_____	Value_____
\$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..79 => 'A', 80 => '1')
\$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..79 => 'A', 80 => '2')
\$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..40=>'A',41=>'1',42..80=>'A')
\$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..40=>'A',41=>'2',42..80=>'A')
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..77=>'0')&"298"

# TEST PARAMETERS

Name_and_Meaning_____	Value_____
<b>*BIG_REAL_LIT</b> A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..74=>'0')&"69.0E1"
<b>*BIG_STRING1</b> A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1=>'"',2..41=>'A',42=>'\"')
<b>*BIG_STRING2</b> A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1=>'"',2..40=>'A',41=>'1',42=>'\"')
<b>*BLANKS</b> A sequence of blanks twenty characters less than the size of the maximum line length.	(1..60 => ' ')
<b>*COUNT_LAST</b> A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
<b>*FIELD_LAST</b> A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
<b>*FILE_NAME_WITH_BAD_CHARS</b> An external file name that either contains invalid characters or is too long.	abc!@def.dat
<b>*FILE_NAME_WITH_WILD_CARD_CHAR</b> An external file name that either contains a wild card character or is too long.	abc*def.dat
<b>*GREATER_THAN_DURATION</b> A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	0.0

# TEST PARAMETERS

Name and Meaning-----	Value-----
#GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	200_000.0
#ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	x#!yz.dat
#ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	(1..60 => 'A')
#INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
#INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
#INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
#LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-0.0
#LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-200_000.0
#MAX_DIGITS Maximum digits supported for floating-point types.	33
#MAX_IN_LEN Maximum input line length permitted by the implementation.	80
#MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
#MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648

## TEST PARAMETERS

Name and Meaning-----	Value-----
<b>#MAX_LEN_INT_BASED_LITERAL</b> A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"2:"&(3..77=>'0')&"11:"
<b>#MAX_LEN_REAL_BASED_LITERAL</b> A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	"16:"&(4..76=>'0')&"F.E:"
<b>#MAX_STRING_LITERAL</b> A string literal of size MAX_IN_LEN, including the quote characters.	(1=>'"',2..79=>'A',80 =>'")
<b>#MIN_INT</b> A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
<b>#NAME</b> A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	SHORT_SHORT_INTEGER
<b>#NEG_BASED_INT</b> A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFF#

## APPENDIX D

### WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- . B28003A: A basic declaration (line 36) incorrectly follows a later declaration.
- . E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);". The Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.
- . C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT\_ERROR.
- . C35502P: The equality operators in line 62 and 69 should be inequality operators.
- . A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT\_ERROR, for that value lies outside of the actual range of the type.
- . C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT\_ERROR, because its upper bound exceeds that of the type.
- . C35904B: The subtype declaration that is expected to raise CONSTRAINT\_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may in fact raise NUMERIC\_ERROR for reasons not anticipated by the test.
- . C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.



## WITHDRAWN TESTS

- . C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.
- . C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT\_ERROR.
- . C37215C, C37215E, C37215G and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.
- . C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT\_ERROR.
- . C41402A: The attribute 'STORAGE\_SIZE is incorrectly applied to an object of an access type.
- . C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE\_OVERFLOW is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE\_OVERFLOW may still be TRUE.
- . C45614C: The function call of IDENT\_INT in line 15 uses an argument of the wrong type.
- . A74106C, C85018B, C87B04B and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT\_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.
- . BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.
- . AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT\_ERROR for implementations which select INT'SIZE to be 16 or greater.
- . CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.
- . CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN\_FILE raises NAME\_ERROR or USE\_ERROR; by Commentary AI-00048, MODE\_ERROR should be raised.

**SUPPLEMENTARY**

**INFORMATION**

AD - A 204279

Ada Compiler Validation Summary Report:

Compiler Name: SYSTEAM Ada Compiler VAX/VMS  
Compiler Version: Version 1.8

Certificate Number: 880524I1.09118

Host and target: VAX 8530 under VMS, Version 4.7

Testing Completed 88-05-24 Using ACVC 1.9

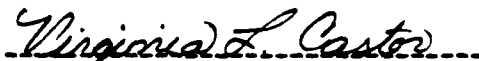
This report has been reviewed and is approved.



IABG m.b.H., Dept SZT  
Dr. H. Hummel  
Einsteinstrasse 20  
8012 Ottobrunn  
Federal Republic of Germany



Ada Validation Organization  
Dr. John F. Kramer  
Institute for Defense Analyses  
Alexandria VA 22311



Ada Joint Program Office  
Virginia L. Castor  
Director  
Department of Defense  
Washington DC 20301